AMD Geode™ GX Processor Based Systems VSA2 Programmer's Guide

March 2006

Publication ID: 32664B

Contacts

www.amd.com

Trademarks

Contents

List of Tables

1 Overview

1.1 Scope

This document describes programming details relevant to AMD's second generation of Virtual System Architecture™ (VSA) technology (hereafter referred to as VSA2) for systems based on the AMD Geode™ GX processor, and the AMD Geode™ CS5535 companion device (hereafter referred to as Geode processor and Geode companion device).

The targeted audience is software engineers who need to:

- Port a system BIOS to VSA2 technology.
- Implement platform-specific System Management Mode (SMM) features via a Virtual Support Module (VSM).

For precise details about structure definitions and macro usage, the programmer should refer to the VSA2.H file.

1.2 Acronyms

ACPI Advanced Configuration and Power Interface

API Application Program Interface

APM Advanced Power Management

ASMI Asynchronous System Management Interrupt

BIOS Basic I/O System

FPU Floating Point Unit

GLIU GeodeLink™ Interface Unit

MSR Model Specific Register

OEM Original Equipment Manufacturer

PM Power Management

RTC Real-time Clock

SMI System Management Interrupt

SMM System Management Mode

SSMI Synchronous System Management Interrupt

VSA2 Virtual System Architecture™ technology

VSM Virtual Support Module

2 Architecture

2.1 VSA2 Architecture

From a VSM programmer's point of view, the main architectural features of VSA2 technology are:

- Client-Server Model
- Virtual Support Modules
- Virtual Registers
- Resource Management

2.1.1 Client-Server Model

The principal architectural feature of VSA2 technology is the client-server model. The advantages of the client-server model are well documented. In the context of VSA2 technology, its usage serves two major purposes. First, it formalizes the interface between modules, thereby creating well defined (thus maintainable) interaction between components. Secondly, it isolates each module from hardware dependencies or variations in system configuration.

The distinction between a client and a server is not per module. Any module can be a client, a server, or both. In the client-server model, each VSM is the server for its particular functionality. If a VSM performs a system call, the VSM is the client and the system manager is the server.

The benefit of the client-server design is demonstrated when the power management (PM) VSM attempts to power down the system. In a monolithic design, this requires the PM module to have knowledge of all of the installed hardware and software components. This would be difficult considering the multitude of hardware configurations and software features. The PM VSM (the client in this example) broadcasts a message to all VSMs, instructing them to set their respective device to a designated power state. This might require the VSMs to unregister for events that must be idle for that power setting. For example, a VSM would typically not leave EVENT_TIMER registered during Standby, since the timer SMIs would awaken the system prematurely.

With this model, the PM VSM is not required to have knowledge of any particular device (SoftVG, USB, SuperI/O, battery controller, etc.) in order to accomplish power management. It only need provide generic PM functionality (legacy timers and system board PM). Each VSM provides the PM functionality specific to the device it controls. Thus, the client-server model facilitates the handling of a variety of platform configurations.

2.1.1.1 System Manager

VSA2 technology may be thought of as the SMM operating system, and the system manager is the kernel of that operating system. The system manager handles the housekeeping functions, serves as the VSM scheduler and dispatcher, and provides the API for the VSMs.

When an SMI occurs, the CPU microcode stores a minimal amount of the processor's state into a SMM header at the physical address determined by the MSR_SMM_HDR MSR. The processor is placed into real mode and execution is vectored to the physical address determined by the MSR_SMM_LOC MSR. The VSA2 initialization code writes these MSRs to the location of the system manager header and entry point respectively. The code at the entry point to the system manager:

- Saves the processor state not already saved by the processor microcode.
- Initializes the processor to the VSA2 technology environment.
- Determines the source(s) of the SMI and enters the event(s) into message queue(s).
- Dispatches to the VSMs that have non-empty message queues.
- Restores the interrupted task's state and resumes execution.

Note: The above implementation is slightly different for nested SMIs.

In addition to the functions mentioned above, the system manager also provides services to VSMs, such as:

- Delivers inter-module messages.
- Abstracts chipset and/or processor dependencies.
- Ensures proper ordering of event handling.
- Implements system calls.
- Saves and restores VSM states as control is switched among VSMs.

Note: A feature new to VSA2 technology for AMD Geode™ processor systems, the floating-point state is saved and restored automatically if (and only if) the FPU is used by a VSM. No special consideration by the VSM is required. A VSM may use the FPU as if it has exclusive ownership. If the FPU is not used, there is no performance impact.

- Provides exception handing.

2.1.1.2 Event Messages

Messaging is the method by which the client requests the server to handle an event. Typically, that event is the result of an SMI, but it could be a system call or virtual register access by a VSM.

All event notification is implemented via a message delivered by the system manager. Therefore, the topmost routine of each VSM consists of a message handling loop. A VSM performs functionality only in response to a message from the system manager. The supported messages are enumerated in Section Section 4.0 "Messages" on page 17.

The message queue for a VSM is stored in the VSM's header. The SYS_GET_NEXT_MSG macro is provided for accessing the message queue. When the SYS_GET_NEXT_MSG macro is invoked and the message queue is empty, control automatically returns to the system manager. A VSM is given control again when there is a message in its queue.

2.1.1.3 System Calls

System calls are provided for requesting services from the system manager. The most common uses of system calls are associated with registering events and handling messages. Some system call functionality could be directly performed by a VSM, however, this would be at the expense of one of the goals of VSA2 technology, which is abstraction of implementation details.

For example, a VSM could write directly to a Geode™ companion device register to enable a feature. However, it is a better design to allow the system manager to perform these functions. If a hardware issue is found requiring a software workaround, the appropriate software changes can be implemented in the system manager and be made transparent to the VSM. Similarly, differences between the different companion device versions can be abstracted, so VSMs can be written to be largely chipset independent. The slight loss in performance due to system overhead is more than made up by readability, portability, and the automatic handling of shared resources. Typically, most chipset programming is performed at initialization, where performance is not critical.

The supported system calls are listed in Section 3.0 "System Calls" on page 9.

2.1.2 Virtual Support Modules

The next architectural feature of VSA2 technology is the partitioning of functionality among VSMs. In most cases, the functionality of a VSM is associated with a particular device, such as video (SoftVG VSM) or USB (OHCI and 8042 VSMs). In other cases, the division is based on a theme, such as power management. In power management, the functionality naturally divides itself along independent modules such as Legacy PM, APM, ACPI, and battery controller. In this example, the desired functionality is accomplished by cooperation between multiple VSMs.

Each major functional component is implemented via a separately compiled VSM. This design allows the ROM builders to concatenate modules with the desired functionality into a customized VSA image. For example, to add APM functionality, the ROM builder simply includes the APM VSM into the VSA2 image. Furthermore, if functionality is dependent on a platform-specific feature (e.g., custom battery controller), the BIOS vendor or OEM has the freedom to implement these details in a VSM without requiring support from the AMD software staff.

2.1.2.1 VSM Structure

A VSM consists of three major components: a VSM header, a VSA2 library, and a message handler. The first two components are supplied by AMD. The message handler is the code that determines the functionality of a VSM.

While VSA2 technology architecturally supports any memory model, it is not anticipated that a VSM would require anything more complex than a single segment for code, data, and stack. This is basically the ‘tiny’ memory model except that segment limits are set to the actual module size instead of 64K.

The first part of every VSM image is a VSM header. The VSM header encapsulates all system data that is specific to a VSM. This design frees the system manager from having to size its data structures based on a maximum number of supported VSMs. Embedded within the VSM header is an SMM header. This header contains most of the VSM's context when it is not executing. The remainder of the VSM's state is saved on its stack. The format of a VSM header is described in VSA2.H.

Each VSM must be linked to a VSA2 technology library file. This library contains the code for the system calls that comprise the application programming interface to the system manager. Since the system code is supplied as a .LIB file, only those object files corresponding to the system calls used by a VSM are linked to the VSM image.

2.1.2.2 VSM Initialization

Typically, only a couple of fields in the sample VSM header file are modified by a VSM writer. The VSM_Type, VSM_Version, and EntryPoint fields are unique to each VSM. The default values for the other fields are usually sufficient for most applications. The remaining fields are initialized by VSA2 software installation code.

The DS, GS and SS descriptors contained in the VSM header are initialized by the VSM install code. When the install code allocates memory of the required size, it initializes the descriptor ‘base’ field and ‘limit’ field (determined from the DS_Limit field). Immediately prior to dispatching to a VSM, the system manager loads the DS, GS, and SS descriptors for that VSM. The FS and ES descriptors are defined as 4 GB flat segments.

Note: A VSM may not modify any of its descriptors.

At VSA2 technology initialization, the system manager puts a MSG_INITIALIZE message into each of the VSM’s message queues. In response to this message, the VSM may perform its initialization, including, but not limited to:

- Data initialization
- Registering events with the system manager
- Configuring hardware

At the end of BIOS POST, the system manager puts another MSG_INITIALIZE message into each of the VSM’s message queues (this message is distinguished from the first one by one of the message parameters). This message gives each VSM the opportunity to perform initialization that may not have been appropriate early in POST.

2.1.2.3 Message Handler

The VSM's message handler implements the functionality of the VSM in response to messages sent by the system manager. It typically consists of an infinite loop of getting a message, parsing the message and any parameter(s) associated with the message, and dispatching to routines to handle the messages. When there are no more messages waiting for the VSM, the system call that retrieves a message returns control to the system manager. The VSM will be dormant until some event occurs that results in a message being entered into its message queue.

A sample message handler is found in Section A.2 "Sample VSM Message Handler" on page 33.

2.1.3 Virtual Registers

Virtual registers are I/O registers accessed with 16-bit index/data pairs. The upper eight bits of a virtual register index specifies a class (group) of virtual registers. The lower eight bits of the index identifies a specific register within the class.

A GLIU descriptor is used to generate an SMI upon access to one of these registers. The system manager determines which VSM has registered for the accessed class, inserts an EVENT_VIRTUAL_REGISTER message into that VSM’s message queue, and schedules the VSM for execution. The VSM’s message handler retrieves the message. The virtual register value, passed as a message parameter, is interpreted by the VSM to perform some appropriate function. The system manager handles the VRC_MISCELLANEOUS class of virtual registers.

Virtual registers are used in two ways. First, they are a uniform method for the BIOS to communicate with VSA2 software, either to pass parameters (e.g., PM timeouts) or implement functionality (e.g., APM). Secondly, virtual registers are a mechanism by which one VSM may communicate with another VSM.

The effects of accessing the virtual register should appear to the VSM to occur immediately, just as it would if the register were a hardware-based I/O register. The scheduling of VSM execution to make this happen is handled by the system manager. To guarantee correct functionality, however, a VSM must use the provided macros (see Section 6.2 "SET_VIRTUAL_REGISTER" on page 25 and Section 6.3 "GET_VIRTUAL_REGISTER" on page 26) for accessing virtual registers.

2.1.4 Resource Management

There are many hardware resources within the SMM environment. Since there are multiple modules that need access to the same resources, it is important that these resources be carefully managed. To do otherwise invites trouble, such as race conditions or spurious interactions. Therefore, the VSA2 architecture carefully partitions responsibility for hardware resources. This is primarily realized in the area of event registration, whereby resources are abstracted from hardware-specific settings to generic events.

2.1.4.1 Event Registration

A VSM may register as a handler of SMI event(s). A list of events is found in Section A.1 "Events" on page 33. Event registration normally occurs at VSM initialization, but dynamic registration is also allowed. Multiple VSMs may register for the same event. The order that VSMs are invoked is determined by the priority specified at the time of event registration. When the event occurs, the VSM that registered for the event with the highest priority is invoked first.

The architecture supports the assignment of a 16-bit priority to each registered event. These priorities are used to control the order that an event is serviced. The eight MSBs specify the order that an SMI is processed relative to other VSMs. The eight LSBs specify the order that an SMI event message is inserted relative to other messages within the VSM's message queue. Thus, the architecture provides for inter-VSM and intra-VSM event prioritization.

For example, assume VSM X registers an event with priority 1000h and VSM Y registers an event with priority 2000h. If VSM X's event is being processed and VSM Y's event occurs, then VSM Y preempts VSM X. VSM X is not given control until the next message in VSM Y's queue is lower priority than the priority of the message being handled by VSM X.

As stated earlier, the eight LSBs control the order that an SMI event message is inserted into a VSM's message queue. A value of 00h in the eight LSBs means the event message will be inserted at the end of the queue. If an event message is already pending in the queue with the same priority as a new event, the newer message is inserted after the older message. That is, a VSM's event queue sorts with the first key being priority and the second key being time of event.

The system manager exclusively handles all SMI-related MSRs. A VSM must not enable/disable any SMI source register nor clear any SMI status bits. If a VSM requires notification of an SMI, it must be implemented by registering for that event.

Note: If a VSM must write to a global resource (e.g., an MSR), the VSM must use read-modify-write techniques to avoid changing any resource state that the VSM does not own exclusively. Since this is likely to be a non-atomic operation (e.g., a RDMSR followed by a WRMSR), the VSM should use the critical section macros to ensure atomicity of the resource modification.

2.1.4.2 Nested SMIs

This architecture supports multi-tasking of VSMs. VSA2 technology makes full use of nested SMIs. That is, nesting is enabled at all times, with the following exceptions:

- Within the system manager
- Within a critical section of a VSM

The effect of nested SMIs is transparent to a VSM. If an asynchronous nested SMI occurs, the VSM that is currently executing is interrupted (its state is saved) and the system manager takes control. The SMI source(s) are determined and message(s) are put into the message queues of the appropriate VSM(s). The interrupted VSM's state is restored and it resumes execution. Except for the small time-slice used by the system manager, the interrupted VSM is unaffected.

As stated above, control is normally returned to the interrupted VSM immediately after handling a nested SMI. There are two exceptions to this rule:

1) Event Priorities: When a VSM registers an event, it assigns a priority to that event. If a high priority event occurs while servicing a lower priority event, the VSM that handles the high priority event is dispatched immediately. When it completes, the VSM handling the lower priority event resumes execution. In the interest of fairness and simplicity, this scenario is only used for events that truly require immediate service (e.g., asynchronous audio SMIs).
2) Virtual Register Access: A VSM that accesses a virtual register expects the effects of the virtual register to occur immediately. In order for this requirement to be satisfied, the VSM that handles the accessed virtual register must preempt the VSM that accessed the virtual register. An EVENT_VIRTUAL_REGISTER message is inserted into the message queue of the VSM that handles the virtual register class that was accessed. The system manager dispatches it to that VSM. When the VSM completes handling of the virtual register, control is returned to the VSM that accessed the virtual register. This scheduling is handled transparently by the system manager.

3 System Calls

There are no system call functions directly accessible to the programmer. System calls are always invoked by a corresponding macro. This provides the flexibility to change the implementation of a macro without changing the programming interface. At a minimum, a VSM must invoke SYS_GET_NEXT_MSG to retrieve messages. Most VSMs also invoke SYS_REGISTER_EVENT one or more times, typically at initialization time. The other system calls are used in specialized cases, as documented in this section.

3.1 SYS_GET_NEXT_MSG

Parameter0: unsigned long * Parameters;

The SYS_GET_NEXT_MSG system call retrieves the next message from a VSM's message queue. The macro returns as a value of type MSG, which is the message code. The macro takes one argument, the address of a buffer to receive the message parameters.

If the VSM's message queue is empty, the macro returns control to the system manager. The VSM regains control again when there is a new message in its message queue.

Example of usage:

unsigned short Message;
unsigned long Parameters[MAX_MSG_PARAM];

Message = SYS_GET_NEXT_MSG(&Parameters);

3.2 SYS_REGISTER_EVENT

Parameter0: unsigned short Event
Parameter1: unsigned long Param1
Parameter2: unsigned long Param2
Parameter3: unsigned long Priority

The SYS_REGISTER_EVENT system call is used to control a VSM's access to the specified event. The parameters passed are dependent on the event being registered. If multiple VSMs register for the same event, the Priority parameter determines which VSM gets the message first.

Unless a VSM is written only to perform some action at initialization, with no run-time processing requirements, it must invoke SYS_REGISTER_EVENT to request notification of desired events. Otherwise, the only events the VSM will receive are those broadcast by the system manager or by other VSMs.

The meanings of the parameters passed to SYS_REGISTER_EVENT are dependent on which event is being registered.

Event Code	Parameter1	Parameter2
EVENT_TIMER	Interval (milliseconds)	16 MSBs = Flags [1] 16 LSBs = Handle
EVENT_IO_TRAP	16 MSBs = Reserved 16 LSBs = I/O base	16 MSBs = Flags [1, 2, 3] 16 LSBs = Range (bytes)
EVENT_IO_TIMEOUT	16 MSBs = Interval (sec.) 16 LSBs = I/O base	16 MSBs = Flags [1, 2] 16 LSBs = Range (bytes)

Event Code	Parameter1	Parameter2
EVENT_DEVICE_TIMEOUT	16 MSBs = Interval (sec.) 16 LSBs = GLIU ID	16 MSBs = Flags[1, 2] 16 LSBs = Instance[4]
EVENT_GPIO[5]	16 MSBs = GPIO Int. # 16 LSBs = GPIO pin #	16 LSBs = Flags
EVENT_PME[6]	16 MSBs = PME# 16 LSBs = GPIO pin #	16 MSBs = PM1 bit 16 LSBs = Flags
EVENT_SOFTWARE_SMI	S/W SMI code (AX)	Mask for AL (e.g., FFh = xx)
EVENT_PCI_TRAP	PCI Address	16 MSBs = Flags[2] 16 LSBs = Address mask
EVENT_VIRTUAL_REGISTER	Virtual register Class	Bits [15:8] - Start Index Bits [7:0] - End Index 0000h for entire Class
EVENT_ACPI	NA	16 MSBs = Flags[1, 2]
EVENT_GRAPHICS	NA	NA
EVENT_KEL	NA	NA
EVENT_USB	NA	NA
EVENT_A20	NA	NA

1. A Flags value of ONE_SHOT may be specified. The event is registered until the occurrence of the event. When the event occurs, it is automatically unregistered.
2. A Flags value of either WRITES_ONLY or READS_ONLY may be specified. If specified, then a message is sent only for I/O accesses of the specified type. If neither WRITES_ONLY nor READS_ONLY is specified, both I/O reads and writes are trapped.
3. If a particular GLIU is to be excluded from use with this registration, one or two (but not all three) of NOT_GLIU0, NOT_GLIU1, NOT_GLIU2 may be specified.
4. The Instance parameter is used when there are multiple devices with the same GLIU Device ID. Normally this parameter is 0x0001.
5. Flags may be one or more of the following: FALLING_EDGE, RISING_EDGE, BOTH_EDGES, DEBOUNCE, PULLUP, PULLDOWN, INVERT, INPUT, OUTPUT, OPEN_DRAIN, AUX1, or AUX2.
6. Flags may be one or more of the following: FALLING_EDGE, RISING_EDGE, BOTH_EDGES, DEBOUNCE, PULLUP, PULLDOWN, INVERT, OPEN_DRAIN, INPUT, AUX1, AUX2, PM1, or GPE.
   If PM1 is set, the 16 MSBs of Parameter2 specifies the bit number to set in the ACPI register PM1_EN. If GPE is set, the bit in the ACPI register PM1_EN corresponding to the specified PME is enabled.

Examples of usage:

```
// Trap writes to primary IDE command register
SYS_REGISTER_EVENT (EVENT_IO_TRAP, 0x1F7, WRITES_ONLY | 1, Priority);

// Register for a 10 millisecond one-shot timer with a handle of 0x1234
SYS_REGISTER_EVENT (EVENT_TIMER, 10, ONE_SHOT | 0x1234, Priority);

// Register for a rising edge on GPIO6
SYS_REGISTER_EVENT (EVENT_GPIO, 6, RISING_EDGE, Priority);

// Register for both edges of GPIO6 to be routed to GPIO PME5
SYS_REGISTER_EVENT (EVENT_PME, 0x00050006, BOTH_EDGES, Priority);

// Register for accesses to PCI addresses 0x80007810-0x8000781F
SYS_REGISTER_EVENT (EVENT_PCI_TRAP, 0x80007810, 0x0000000F, Priority);
```

3.3 SYS_UNREGISTER_EVENT

Parameter0: unsigned short Event
Parameter1: unsigned long Param1
Parameter2: unsigned long Param2

The SYS_UNREGISTER_EVENT system call cancels a prior event registration. The parameters passed should be the same values used when the event was registered.

3.4 SYS_VSM_PRESENT

Parameter0: unsigned char VSM_Type

The SYS_VSM_PRESENT system call checks for the presence of a VSM of the specified type. If present, a value of TRUE is returned, otherwise FALSE. A VSM may use this macro to modify its behavior based on the presence of another VSM. For example, if an APM VSM is present, the PMCore VSM may report an APM event when the Standby button is pressed. If the APM VSM is not present, then the PMCore VSM handles the button event.

3.5 SYS_UNLOAD_VSM

No Parameters

The SYS_UNLOAD_VSM system call removes the calling VSM from VSA2. This macro is typically used if a VSM determines that its functionality is not required in the current system. For example, an OEM may support multiple platforms with a variety of SuperI/O chips. The VSA2 image may include SuperI/O VSMs for each supported chip. In response to the first MSG_INITIALIZE, each SuperI/O VSM would determine if its chip is present in the system. If not, the VSM can remove itself from the runtime VSA2 image.

3.6 SYS_LOGICAL_TO_PHYSICAL

Parameter0: void * Address;

The SYS_LOGICAL_TO_PHYSICAL system call returns the physical address of a local variable in a VSM. This is typically useful for VSMs that must supply a physical address to a bus-mastering device.

3.7 SYS_RETURN_RESULT

The SYS_RETURN_RESULT system call is used in response to an EVENT_IO_TRAP message to return an emulated value. The macro determines the appropriate operand size (BYTE, WORD, or DWORD) as well as the proper destination: EAX, AX, or AL for IN instructions or ES:[DI] for INS instructions.

3.8 SYS_YIELD_CONTROL

Parameter0: unsigned long Milliseconds

The SYS_YIELD_CONTROL system call suspends execution of a VSM for the specified number of milliseconds. This would typically be used in cases where a VSM needs to wait on a device or requires a delay. It can be disruptive to some time-sensitive applications and/or device drivers to delay for too long in SMM. If a VSM needs to wait for some status that may take more than about 100 microseconds, it is better to yield control to give the non-SMM code time to execute. When the requested interval has elapsed, control is returned to the requesting VSM.

Note: This macro yields control even if the VSM's message queue has a message in it. However, if a VSM has yielded control via this system call, any subsequent event belonging to the VSM will awaken it, even if the requested time has not elapsed.

Example of usage:

```
// Yield control for 5 ms
SYS_YIELD_CONTROL(5);
```

3.9 SYS_BROADCAST_MSG

Parameter0: unsigned short Message
Parameter1: void * Parameter_Buffer
Parameter2: unsigned short VSM_Type

The SYS_BROADCAST_MSG system call inserts a message into the message queue of one or more VSMs.

The requesting VSM also receives the broadcast message, except as noted below in *Parameter2*. The receipt of this message by the broadcasting VSM indicates all VSM(s) targeted by the message have received it and have acted on it. The VSM scheduler in the system manager ensures that this synchronizing message is sent only after the receiving VSM(s) have executed in response to this message.

This system call is typically used to notify VSMs of an important change in the state of the system such as power state or an imminent warm/cold boot. Another important usage is to instruct VSMs to save state in preparation for a Save-to-RAM.

- *Parameter0* specifies the message to be entered.
- *Parameter1* specifies the message parameters (if any).
- *Parameter2* specifies the type of VSM into whose message queue the message is inserted. If there are multiple VSMs of that type, all VSMs of that type receive the message. A special VSM type, VSM_ANY, specifies that the message is to be sent to all VSMs. One of two flags may be optionally ORed with the VSM type parameter:
  — VSM_NOT_SELF signals that the message is not to be sent to the broadcasting VSM.
  — VSM_ALL_EXCEPT signals that the message is to be sent to all VSMs except the one specified. Use of VSM_ALL_EXCEPT with VSM_ANY is an invalid combination.

Examples of usage:

```
// Instruct all VSMs to save the state of their associated hardware.
// The sending VSM call receives the message too.
SYS_BROADCAST_MSG(MSG_SAVE_STATE, &ParamsBuffer, VSM_ANY);

// Notifies all VSMs of a change in the PM mode.
// The message is not sent to the sending VSM (it already knows the mode is changing)
SYS_BROADCAST_MSG(MSG_SET_POWER_MODE, &PowerMode, VSM_NOT_SELF);

// An event message is sent to all VSMs except the APM VSM.
SYS_BROADCAST_MSG(MSG_EVENT, &ParamsBuffer, VSM_ALL_EXCEPT | VSM_APM);
```

3.10 SYS_SW_INTERRUPT

Parameter0: unsigned short INT_Number
Parameter1: Regs * Registers

The SYS_SW_INTERRUPT system call performs a software interrupt. The interrupt vectors used are the ones set up by the BIOS and/or option ROMs that existed just before the end of POST (system manager takes a snapshot of the interrupt vectors at the end of POST).

Note: This system call is supplied primarily to support video power management through VESA calls (INT 10h) and for Save-to-Disk (INT 13h). Some interrupts may not work, depending on how “well-behaved” they are.

The only interrupt supported before end of POST is INT 15h, since it has a standard vector. This implies that SYS_SW_INTERRUPT may not be used in response to the early MSG_INITIALIZE.

The structure passed to this system call contains the IRQ masks to be used during the software interrupt. Typically, all IRQs are masked except for ones associated with the specified software interrupt (e.g., INT 13h). The interrupt vector used for the INT call is the one that existed at the end of POST. Any other vectors that might be used during the INT call are the current vectors.

This system call must not be used if the operating system does not preserve the BIOS memory space. If the interrupt vector points to an option ROM, that memory space must also be preserved.

Example of usage:

```
// Perform VESA video call to set display power state to Standby
Regs Registers;                 // The Regs structure is defined in VSA2.H
Registers.Reg_AX = 0x4F10;      // Major function code
Registers.Reg_BL = 0x01;        // Minor function code (Set Display Power State)
Registers.Reg_BH = 0x01;        // Power state = Standby
Registers.Reg_CX = 0x0000;      // Some video BIOSes require CX == 0000
Registers.PIC0_Mask = 0x00;     // Mask all h/w IRQs
Registers.PIC1_Mask = 0x00;
SYS_SW_INTERRUPT(0x10, &Registers);
```

3.11 SYS_GET_SYSTEM_INFO

Parameter0: Hardware * SystemInfo

The SYS_GET_SYSTEM_INFO system call is used to get information about the system on which the VSM is currently executing. This is useful in that it allows a VSM to tailor its behavior according to different CPU or chipset versions. Therefore, a single VSM binary is able to execute on a variety of platform configurations.

Example of usage:

```
typedef struct {
    unsigned long Chipset_Base;     // Southbridge's PCI Address
    unsigned short Chipsete_ID;     // Southbridge's PCI Device ID
    unsigned short Chipset_Rev;     // Southbridge's Revision
    unsigned short CPU_ID;          // CPU's PCI Device ID
    unsigned short CPU_Revision;    // CPU's revision
    unsigned short CPU_MHz;         // Units = MHz
    unsigned long SystemMemory;     // Units = bytes
    unsigned long VSA_Location;     // Physical location
    unsigned short VSA_Size;        // Units = KB
    unsigned short PCI_MHz;         // Units = MHz
} Hardware;
Hardware SysInfo;
SYS_GET_SYS_INFO (&SysInfo);
```

3.12 SYS_MAP_IRQ

Parameter0: unsigned char IRQ_Source
Parameter1: unsigned char IRQ

This system call maps an Unrestricted IRQ Source to the specified IRQ/SMI. An IRQ value of 2 specifies that the Y Source event should generate an SMI. An IRQ value of 0 specifies that the IRQ Source event is to be disabled (neither IRQ nor SMI generated). Valid values for IRQ Source are shown below. All other values are reserved.

1 - USB 1

2 - USB 2

3 - RTC Alarm

4 - Audio Codec

5 - Power Management (SCIs)

6 - NAND Flash Ready

7 - NAND Flash Distraction

12 - SMB Controller

13 - KEL Emulation

14 - UART 1

15 - UART 2

Examples of usage:

```
// Map Y Source 3 (RTC alarm) to an SMI
SYS_MAP_IRQ(3, 2);

// Map SCIs to IRQ 10
SYS_MAP_IRQ(5, 10);
```

3.13 Support for Virtualized PCI Headers

A few system calls have been created to facilitate creation of virtualized PCI headers. The system manager must have a pre-defined entry in its table of virtualized PCI headers. These functions are used primarily by the system manager and SoftVG. They are not normally used by OEM-written VSMs.

3.13.1 SYS_ALLOCATE_RESOURCE

Parameter0: unsigned char BAR_Type
Parameter1: unsigned short BAR_Offset
Parameter2: unsigned long Length
Parameter3: unsigned short PCI_DeviceID
Parameter4: unsigned short GeodeLink_DeviceID

This system call is used to create a linkage between a virtualized PCI base address regiser and a GeodeLink™ device. The macro returns the PCI address assigned to the requested resource.

Parameter0 specifies the type of resource to be allocated:

- RESOURCE_MEMORY: Memory BAR will be allocated.
- RESOURCE_MMIO: Memory-mapped BAR is allocated.
- RESOURCE_IO: I/O BAR will be allocated.

Parameter2 specifies the size in bytes of the requested resource. If the resource type is RESOURCE_MEMORY, a block of physical memory is allocated from the end of extended memory.

Taken together, *Parameter1* and *Parameter3* specify the PCI configuration address of the resource to be allocated.

Parameter4 specifies the GeodeLink device to be linked to the resource. The IDs for GeodeLink devices are defined in GX2.H

All MSR(s) appropriate to the device and requested resource type are allocated (RCONF, GLIU descriptor, DD LBAR, etc.). The system manager records the association between virtualized PCI header and MSR(s). Subsequent writes to the virtualized PCI resource (BAR or Command registers) results in the linked (associated) MSR(s) being updated.

Example of usage:

```
// Allocate 8 MB of physical memory to BAR0 for the video frame buffer
SYS_ALLOCATE_RESOURCE(RESOURCE_MEMORY, BAR0, 0x800000, 0x0030, ID_MC);
```

3.13.2 SYS_LOOKUP_DEVICE

Parameter0: unsigned short GeodeLink_Device_ID
Parameter1: unsigned short Instance

This system call searches the GeodeLink topology for the specified GeodeLink device. Use of this macro allows a VSM to accommodate changes in the GeodeLink topology; that is, MSR addresses of devices do not need to be hard-coded.

The IDs for GeodeLink devices are defined in GX2.H. SYS_LOOKUP_DEVICE returns the specified GeodeLink device's routing address. The four LSBs of the return value is the device's port number of the GLIU on which it is located. If the Device ID was not found, 00000000h is returned. The second parameter specifies which instance of the device is to be found (in case there are multiple devices of the same type (e.g., OHCI). Normally this parameter is 0001h.

Example of usage:

```
// Search for the first instance of the OHCI device
// For the AMD Geode™ CS5535 companion device, the value 0x51200002 is returned
GeodeAddress = SYS_LOOKUP_DEVICE(ID_OHCI, 1);
```

3.13.3 SYS_MBUS_DESCRIPTOR

Parameter0: unsigned short PCI_Address;
Parameter1: unsigned long * MsrValue;

This system call is used to determine the GeodeLink descriptor value that is linked to a virtualized PCI BAR.

Note: This system call is not used by a typical VSM. It is included for VSMs that need to extract information about a memory mapped region.

3.13.4 SYS_IO_DESCRIPTOR

Parameter0: unsigned short IO_Address;
Parameter1: unsigned long * MsrValue;

This system call is used to determine the GeodeLink descriptor address that is linked to an I/O range. It is used primarily to allow a power management VSM to disable I/O trapping during critical power state sequences. The system call returns as its value the MSR routing address of the descriptor that maps the specified I/O range. If no mapping exists, the value 0x00000000 is returned. *Parameter1* should point to two consecutive DWORDs to hold the current MSR value. The system call sets the descriptor to its default value.

Note: It is the VSM's responsibility to restore the descriptor to its original value.

Example of usage:

```
ULONG MsrAddr, MsrValue[2];

    // Disable the I/O descriptor routing address and current value
    MsrAddr = SYS_IO_DESCRIPTOR(0x9C00, MsrValue);

        <untrapped I/O accesses to 0x9C00 go here>

    // Restore I/O descriptor
    if (MsrAddr) Write_MSR(MsrAddr, MsrValue);
```

4 Messages

Messages are retrieved from a VSM's message queue with the GET_NEXT_MSG macro. The GET_NEXT_MSG macro takes one parameter, the address of an array of *unsigned longs*. Messages may have zero to MAX_MSG_PARAM parameters associated with them, depending on the message. The GET_NEXT_MSG macro fills the array with the parameters, if any, associated with the retrieved message. The macro returns an *unsigned short*, which is the message code associated with the current message parameters.

Note that a VSM receives most of the following messages at some time during its execution. The only message that a VSM has control over is MSG_EVENT. A VSM receives MSG_EVENT only for events for which it registers. A VSM is free to ignore some messages. For example, a VSM may not have hardware state to save or restore, so it would ignore the MSG_SAVE_STATE and MSG_RESTORE_STATE messages.

4.1 MSG_INITIALIZE

Parameter0: unsigned long Flag

This message is sent to each VSM twice. The first message (Parameter0 = EARLY_INIT) is sent when VSA is first installed (early POST). Examples of the types of initialization performed in early POST are:

- Registration of events
- Hardware initialization
- VSM Initialization

The system BIOS signals the end of POST immediately before booting the operating system. In response to this signal, the system manager sends a second MSG_INITIALIZE message (Parameter0 = END_OF_POST_INIT) to each VSM. This allows each VSM to perform initialization that may not have been appropriate at the earlier initialization.

4.2 MSG_EVENT

Parameter0: Event_Code
Parameter1: Dependent on event
Parameter2: Dependent on event
Parameter3: Dependent on event

MSG_EVENT is, by far, the most common message. When a registered event occurs, this message is sent to the VSM that has registered for that event with the highest priority. If a VSM handles the event, the VSM does not have to do anything explicitly to notify the system manager. However, if a VSM chooses not to handle the event, it should invoke the SYS_PASS_EVENT system call to give other VSMs that have registered for the same event an opportunity to handle the event. If no VSM handles the event (all registered VSMs invoked the SYS_PASS_EVENT system call), an error is signaled.

Table 4-1 is a summary of the parameters passed for each type of MSG_EVENT:

Table 4-1. MSG_EVENT Parameter Summary

Parameter0	Parameter1	Parameter2	Parameter3
EVENT_GRAPHICS EVENT_IO_TRAP EVENT_ACPI EVENT_A20	SMM Header Flags	31:16 - Data Size 15:00 - I/O Address	Data (if I/O write)

Table 4-1. MSG_EVENT Parameter Summary

Parameter0	Parameter1	Parameter2	Parameter3
EVENT_IO_TIMEOUT	NA	15:00 - I/O Address or DeviceID	NA
EVENT_GPIO	GPIO Pin	RISING_EDGE or FALLING_EDGE	NA
EVENT_SOFTWARE_SMI	0000::AX	EBX	ECX
EVENT_PCI_TRAP	PCI Address	Bits 31:16 - Reserved Bit 7 - Set if PCI Write Bits 3:0 - Data Size	Data (if PCI write)
EVENT_VIRTUAL_REGISTER	15:08 - VR Class 07:00 - VR Index	0 = Read 1 = Write	Data (if VR write)
EVENT_TIMER	Interval	15:0 - Handle	NA
EVENT_USB	HC Address	HC Number	NA
EVENT_KEL	HC Address	HC Number	HCE Control
EVENT_PME	NA	ACPI Wake mask	NA

4.3 MSG_SAVE_STATE

No Parameters

This message is sent to each VSM in preparation for a Save-to-RAM. All information that is required for restoring the device(s) controlled by the VSM must be saved to the VSM's data segment. This message is sent at most one time by the Save-to-Disk VSM.

4.4 MSG_RESTORE_STATE

No Parameters

This message is sent to each VSM during a Resume-from-RAM. All devices handled by the VSM should be restored using the state information that was saved during the handling of the MSG_SAVE_STATE message. This message is sent at most one time by the Save-to-RAM VSM, and is only sent if a prior MSG_SAVE_STATE has been issued.

4.5 MSG_SET_POWER_MODE

Parameter0: PowerMode
Parameter1: APM connect status (only if PowerMode = PM_APM)

This message is sent to each VSM when the power management mode is changed. This message may be sent by any VSM involved in power management.

Table 4-2. Valid Power Modes

Value	Description
PM_DISABLED	Power management is completely disabled
PM_LEGACY	Power management is controlled by inactivity timers
PM_APM	Power management is controlled by APM driver
PM_ACPI	Power management is controlled by operating system and ASL code

4.6 MSG_SET_POWER_STATE

Parameter0: PowerState

This message is sent to each VSM when the power level of the device(s) controlled by the VSM is to be changed. For example, before the system enters Standby, all VSMs are sent the MSG_SET_POWER_STATE message with Parameter0 set to S1_STATE.

Table 4-3. Valid Power States

Value	Description
S0_STATE	Full ON
S1_STATE	CPU and RAM context preserved; CPU clock stopped
S2_STATE	CPU and RAM context preserved; most clocks stopped (Not used by VSA)
S3_STATE	RAM context preserved; CPU context saved and CPU is powered off
S4_STATE	RAM and CPU contexts saved on disk; system is powered off (Not used by VSA)
S5_STATE	Soft OFF

4.7 MSG_ABORT_POWER_STATE

No Parameters

Upon receipt of a MSG_SET_POWER_STATE message, a VSM may broadcast this message to indicate that it wants to abort the new power state. Note that some VSMs may have already acted on the previous MSG_SET_POWER_STATE message. This means that the controlling VSM (typically the VSM that sent the original message) should broadcast a MSG_SET_POWER_STATE message to return the power state to an appropriate setting (e.g., S0_STATE).

4.8 MSG_WARM_BOOT

No Parameters

This message is sent to each VSM immediately before the system is warm booted. VSMs should respond to this message by preparing their specific hardware for the warm boot. For example, any VSMs that control a bus master device should ensure any bus mastering is completed. In general, VSMs do not need to unregister from all events. They need to disable hardware that is unknown to the system manager. The system manager disables all appropriate Southbridge functionality.

4.9 MSG_SHUTDOWN

No Parameters

This message is sent to each VSM immediately before the system is reset. It provides the opportunity to shut the system hardware down gracefully by disabling hardware or waiting for certain activity to complete. In particular, all SMI sources should be disabled. Processing for this event is typically the same as that for MSG_WARM_BOOT.

4.10 MSG_QUEUE_OVERFLOW

Parameter0: Event code (see Section A.1 "Events" on page 33)

The system manager always reserves one message queue entry for this message. If an event occurs that overflows the message queue, a MSG_QUEUE_OVERFLOW message is sent instead of the original message. The original message code is passed in *Parameter0*. If any more events occur before the MSG_QUEUE_OVERFLOW message is retrieved, those events are discarded and an error is recorded. This error message can be retrieved with the INFO utility.

This message is primarily designed to notify a VSM of event overrun. In many cases, nothing can be done at runtime about the situation. This message's intent is to notify the VSM writer that a serious problem exists. It may be that a hardware device is generating SMIs too fast, or that unintentional delays exist in the VSM such that it cannot service the message queue quickly enough.

5 Event Messages

The main routine of all VSMs is a message-handling loop. When an event occurs, the VSMs that have registered for that event are sent a MSG_EVENT message with the event code in *Parameter0*. Other parameters may also be passed, depending on the event. A VSM is only sent a MSG_EVENT message for the events that it registered as a handler.

5.1 EVENT_TIMER

Parameter0: EVENT_TIMER
Parameter1: Interval
Parameter2: Handle

A VSM may request to be notified every n milliseconds. This event is generated when the requested time interval has expired. The timer request remains active until the VSM explicitly cancels the request, via SYS_UNREGISTER_EVENT.

Since a VSM may request more than one timer to run concurrently, a unique handle (to the VSM, not system-wide) may be specified when an EVENT_TIMER is registered. The handle must be less than or equal to FFFFh. An optional flag (ONE_SHOT) may be ORed with the handle parameter to generate a timer that is automatically unregistered upon expiration of the interval. This saves the programmer from having to perform:

SYS_UNREGISTER_EVENT (EVENT_TIMER, Interval, Handle);

when the EVENT_TIMER message is received.

When a timer elapses, the corresponding handle is passed as a parameter of the EVENT_MSG to allow the VSM to distinguish among multiple concurrent timers.

5.2 EVENT_GRAPHICS

Parameter0: EVENT_GRAPHICS
Parameter1: Flags field (from SMM header)
Parameter2: [31:16] = Data Size field from SMM header
[15:0] = I/O address from SMM header
Parameter3: Data (if I/O write)

This event is sent when there is an SMI from the graphics unit. The parameters are only valid for trapped I/O, not asynchronous graphics events.

5.3 EVENT_IO_TRAP

Parameter0: EVENT_IO_TRAP
Parameter1: Flags field (from SMM header)
Parameter2: [31:16] = Data Size field from SMM header
[15:0] = I/O address from SMM header
Parameter3: Data (if I/O write)

An access to a registered I/O location has occurred.

5.4 EVENT_IO_TIMEOUT

Parameter0: EVENT_IO_TIMEOUT
Parameter1: I/O address or DeviceID

An access to a monitored I/O range (or ranges if Device ID was specified) has not occurred for the interval specified when this event was registered.

5.5 EVENT_GPIO

Parameter0: EVENT_GPIO
Parameter1: GPIO pin number
Parameter2: Flags

A transition has occurred on a GPIO pin. The following symbols are defined for the programmer's convenience:

GPIO_<n>, where the maximum value of <n> varies depending on the Geode™ companion device in use.

5.6 EVENT_SOFTWARE_SMI

Parameter0: EVENT_SOFTWARE_SMI
Parameter1: SMI code

The receipt of this event signals that a software SMI has occurred. A VSM will only be sent this message for software SMIs of the value for which it registered. The SMI code is the value in AX at the time the SMI occured.

Note: **The BIOS should use virtual registers for communication with VSA.** This event is provided for backward compatibility.

5.7 EVENT_PCI_TRAP

Parameter0: EVENT_PCI_TRAP
Parameter1: PCI Configuration Address
Parameter2: 01h = BYTE access
03h = WORD access
0Fh = DWORD access
Bit 7 set if access is an I/O write
Parameter3: PCI register value

An access to a registered PCI configuration register has occurred.

Note: The Northbridge and Southbridge PCI configuration registers are virtualized on Geode processor systems. The system manager handles this virtualization prior to sending EVENT_PCI_TRAP messages to any registered VSMs. The virtualized value is passed in *Parameter3* on reads. On writes, the value written is passed in *Parameter3*.

5.8 EVENT_VIRTUAL_REGISTER

Parameter0: EVENT_VIRTUAL_REGISTER
Parameter1: Virtual register (Bits [15:8] = VR class; Bits [7:0] = Class index)
Parameter2: 0 = Word read
1 = Word write
All others reserved
Parameter3: Data (valid only if access is a write)

An access to a registered virtual register has occurred.

5.9 EVENT_USB

Parameter0: EVENT_USB
Parameter1: HostControllerAddress
Parameter2: HceControl

A USB event has occurred on the specified OHCI host controller.

5.10 EVENT_KEL

Parameter0: EVENT_KEL
Parameter1: HostControllerAddress
Parameter2: HceControl

A keyboard emulation event has occurred on the specified OHCI host controller.

5.11 EVENT_A20

Parameter0: EVENT_A20

A *change* in the A20 mask has occurred. This may be either by bit 1 of I/O port 92h, or by the issuing of a D1h keyboard controller command.

6 Macros

In addition to the system call macros documented in Section 3.0 on page 9, there are other macros available to the VSM programmer. These are designed to achieve certain required functionality without the burden of understanding the implementation details. For example, if a VSM requires information from the SMM header, it has no way of knowing where the correct SMM header is located. Using the provided macros abstracts these details.

6.1 Critical Sections

The macros ENTER_CRITICAL_SECTION and EXIT_CRITICAL_SECTION are used to mark the beginning and end of VSM code that should not be interrupted. Typically, this would be brief sections of code that are very timing sensitive. For example, if a VSM must create a software-generated pulse of precise duration, the VSM must be ensured it will not be interrupted by some asynchronous event. If this occurred, the actual pulse width would be longer than desired. Bracketing such code with critical section macros prevents any interruption.

The other reason for the need for critical sections is to ensure atomic operations. Atomic operations are required for both hardware (index/data pairs or read-modify-write sequences) and software (data structure consistency). Since VSMs are neither reentrant nor share data structures with other VSMs, data structure consistency is not usually an issue in a VSM.

An example of the need for atomicity is when a VSM accesses an index/data I/O pair, as with an access to the real-time clock (RTC). It is possible for the VSM to be interrupted by a high priority asynchronous event after writing the RTC index register but before writing the data register. If the interrupting VSM accesses the same index/data pair, it corrupts the index from the first VSM. In order to prevent this scenario from occurring, such a section of code must be handled as a critical section by bracketing the code with the appropriate critical section macro calls (see example below).

It should be noted that although critical sections ensure atomicity, the VSM programmer must still save and restore index register values unless those registers are owned exclusively by the VSM. The exception to this rule is the PCI configuration address at CF8h. Since PCI configuration space accesses are very typical in VSM coding, this address is considered part of the VSM's context and is saved and restored by the system manager.

Critical sections must be used very carefully. In particular, most system calls do not work properly inside of a critical section, since they depend on the generation of a software SMI to dispatch to the system manager.

Example of usage:

```
ENTER_CRITICAL_SECTION;
< critical code>
EXIT_CRITICAL_SECTION;
```

6.2 SET_VIRTUAL_REGISTER

The SET_VIRTUAL_REGISTER macro is used to write to a virtual register from within a VSM. Accessing virtual registers directly from a VSM (via I/O) is *not permitted.* On an error-checking version of the system manager, an error is logged.

Example of usage:

```
unsigned short ClassIndex;
ClassIndex = VRC_APM << 8 | REPORT_EVENT;
SET_VIRTUAL_REGISTER(ClassIndex, 6);
```

6.3 GET_VIRTUAL_REGISTER

The GET_VIRTUAL_REGISTER macro is used to read to a virtual register from within a VSM. Accessing virtual registers directly from a VSM (via I/O) is *not permitted.* On an error-checking version of the system manager, an error is logged.

Example of usage:

```
unsigned short ClassIndex, SleepPin;
ClassIndex = VRC_PM << 8 | SLEEP_PIN;
SleepPin = GET_VIRTUAL_REGISTER (ClassIndex);
```

6.4 CPU State Access

Macros are provided to retrieve CPU state information. This is typically used when servicing synchronous events such as I/O traps, software SMIs, or virtual registers.

6.5 GET_HEADER_DATA

Parameter0: SMM_Header_Field

The GET_HEADER_DATA macro is used to read data from the SMI header from the interrupted (non-SMM) task. The size of the field retrieved is determined by the field.

Valid SMM header field names are:

```
NEXT_EIP
CURRENT_EIP
CS_LIMIT
CS_BASE
CS_SELECTOR
CS_ATTRIBUTE
EFLAGS
DATA_SIZE
IO_ADDRESS
WRITE_DATA
SMM_FLAGS
R_CR0
R_DR7
```

Example of usage:

```
// Get address of trapped instruction
Address = GET_HEADER_DATA (NEXT_EIP);
```

6.6 SET_HEADER_DATA

Parameter0: SMM_Header_Field

The SET_HEADER_DATA macro is used to write data to the SMI header from the interrupted (non-SMM) task. The size of the field written is determined by the field.

Note: This system call is only valid and meaningful when servicing a synchronous event (EVENT_SOFTWARE_SMI or EVENT_IO_TRAP). An attempt to use it when servicing an asynchronous event is a programming error.

Valid SMM header field names are defined in Section 6.5 "GET_HEADER_DATA".

6.7 SET_REGISTER

Parameter0: Register name
Parameter1: Data value

The SET_REGISTER macro is used to modify the interrupted (non-SMM) task registers.

Note: This system call is only valid and meaningful when servicing a synchronous event (EVENT_SOFTWARE_SMI or EVENT_IO_TRAP). An attempt to use it when servicing an asynchronous event is a programming error.

Valid register names are:

R_EAX, R_AX, R_AH, R_AL
R_EBX, R_BX, R_BH, R_BL
R_ECX, R_CX, R_CH, R_CL
R_EDX, R_DX, R_DH, R_DL
R_ESI, R_SI
R_EDI, R_DI
R_EBP, R_BP
R_ESP, R_SP

Example of usage:

```
SET_REGISTER (R_EAX, 0x123456578);     // Set caller's EAX to 0x12345678
```

6.8 GET_REGISTER

Parameter0: Register name

The GET_REGISTER macro is used to retrieve a general purpose register from the interrupted (non-SMM) task.

Note: This system call is typically only meaningful when servicing a synchronous event (e.g., EVENT_SOFTWARE_SMI, EVENT_IO_TRAP).

Example of usage:

```
Register_EDX = GET_REGISTER (R_EDX);     // Get caller's EDX
```

6.9 Shorthand Macros

Since the AL, AX and EAX registers are the most commonly accessed registers, specific macros are defined to facilitate readability of the code. These macros and their equivalent definitions are:

Table 6-1. Shorthand Macros

Shorthand Macro	Equivalent Macro
GET_AL( )	GET_REGISTER(R_AL)
GET_AX( )	GET_REGISTER(R_AX)
GET_EAX( )	GET_REGISTER(R_EAX)
SET_AL(data)	SET_REGISTER(R_AL, (*unsigned char*) data)
SET_AX(data)	SET_REGISTER(R_AX, (*unsigned short*) data)
SET_EAX(data)	SET_REGISTER(R_EAX, (*unsigned long*) data)

6.10 PCI Configuration Space Access

Misaligned accesses to virtual PCI configuration space are handled correctly by the system manager. However, a VSM should ensure that PCI accesses are naturally aligned in case a VSM traps the virtualized PCI access and does *not* handle misaligned accesses correctly.

6.10.1 READ_PCI_BYTE

Parameter0: PCI Configuration Address

Example of usage: HeaderType = READ_PCI_BYTE (0x8000780E);

6.10.2 READ_PCI_WORD

Parameter0: PCI Configuration Address

Example of usage: DeviceID = READ_PCI_WORD (0x80007802);

6.10.3 READ_PCI_DWORD

Parameter0: PCI Configuration Address

Example of usage: BAR0 = READ_PCI_DWORD (0x80007810);

6.10.4 WRITE_PCI_BYTE

Parameter0: PCI Configuration Address
Parameter1: Data to write

Example of usage: WRITE_PCI_BYTE (0x80007D3D, 0x0A);

6.10.5 WRITE_PCI_WORD

Parameter0: PCI Configuration Address
Parameter1: Data to write

Example of usage: WRITE_PCI_WORD (0x80007D04, 0x0003);

6.10.6 WRITE_PCI_DWORD

Parameter0: PCI Configuration Address
Parameter1: Data to write

Example of usage: WRITE_PCI_DWORD (0x80007914, 0xFFE00000);

6.10.7 PCI NO_TRAP Macros

There are situations where direct access is required to PCI configuration space. That is, a VSM may require access to the underlying hardware PCI space without generating a SSMI. In such cases, the NO_TRAP macros are applicable. A typical example where these macros are useful is when a VSM traps I/O writes to a PCI register range. Since the configuration cycle was trapped, a PCI configuration cycle to the real underlying hardware is intercepted. Typically, the real hardware register needs to be updated, but simply re-issuing the I/O would cause another SSMI. To update the real hardware register, the NO_TRAP macros should be used.

These macros are only useful for accessing PCI configuration space of real hardware devices. Virtualized PCI devices have no underlying hardware configuration space. Therefore, the NO_TRAP macros perform the same functionality as the equivalent macro without the NO_TRAP suffix. Care must be taken by VSMs to avoid trapping a virtualized PCI register and re-issuing the configuration cycle. To do so would cause another EVENT_PCI_TRAP to be sent to the issuing VSM. This creates an infinite sequence of messages. It is not necessary to re-issue configuration cycles to virtualized PCI registers. The NO_TRAP macros are:

READ_PCI_BYTE_NO_TRAP
READ_PCI_WORD_NO_TRAP
READ_PCI_DWORD_NO_TRAP
WRITE_PCI_BYTE_NO_TRAP
WRITE_PCI_WORD_NO_TRAP
WRITE_PCI_DWORD_NO_TRAP

7 VSA2 Utility

The INFO.EXE program is a useful utility when developing a VSM. Several categories of information are available from this VSA2 utility.

7.1 Help Screen

"INFO /?" displays the available switches to access the other information categories. It displays a help screen as shown below.

```
VSA II Utility v3.02
Copyright (c) 1999-2003 by Advanced Micro Devices

Displays information about the current VSA II or a file image.

Usage:  INFO [file.ext] [/R|r] [/S|s] [/H|h] [/E|e]

   /?  displays help screen
   /r  displays registered events
   /s  displays statistics       /S to clear
   /e  displays error log        /E to clear
   /h  displays event history    /H to clear
   /G  displays info about GPIOs
   /D  displays MSRs
       filters: i=I/O m=Memory l=LBAR r=RCONF t=traps f=free
   /W  unlocks write-protect on VSA memory
   /X  GLIU documentation
```

7.2 Basic Information

Executing "INFO" without any switches displays detailed information about each VSM. The first line after the copyright message displays information about the Geode™ processor and companion device. The most important fields, the VSM's base address and length, tell the memory range that the VSM occupies. Together with a .MAP file and the VSM base, a memory editor/viewer can be used to debug a VSM. The IP and SP columns show the respective registers in both offset/absolute formats.

```
VSA II Utility v3.02
Copyright (c) 1999-2003 by Advanced Micro Devices

200 MHz Geode GX Processor 1.0 with CS5535 1.3

  VSM Type   Version   VSM Base     Length          IP             SP
-----------  -------   --------   ----------   -------------   -------------
00 SysMgr     03.73    40400000   A53C 41.3K   0421/40400421   07EC/404007EC
03 Legacy     01.06    4040A540   2420  9.0K   02E8/4040A828   23E8/4040C928
02 SoftVG     01.01    4040C960   5660 21.6K   359A/4040FEFA   562C/40411F8C
05 OHCI       02.21    40411FC0   5D90 23.4K   0360/40412320   5D5A/40417D1A
06 8042       02.21    40417D50   1B70  6.9K   0312/40418062   1B34/40419884
08 ACPI       01.00    404198C0   2570  9.4K   05C4/40419E84   2518/4041BDD8
04 PM         01.30    4041BE30   16A0  5.7K   03A4/4041C1D4   166C/4041D49C
09 APM        01.21    4041D4D0   1F50  7.8K   043C/4041D90C   1F08/4041F3D8
0E Save2Dsk   01.20    4041F420   6980 26.4K   0A26/4041FE46   693C/40425D5C
05 OHCI       02.21    40425DA0   5D90 23.4K   0360/40426100   5D5A/4042BAFA

Total VSA memory allocated:        256.0K
Total VSA memory used:             174.8K
Total VSA memory available:         81.2K
```

7.3 Registered Events

"INFO /r" displays the events for which each VSM is registered. This is useful to confirm that the VSM has registered for the expected events and with the correct parameters. Since a VSM may dynamically register and unregister for events, this display may be different each time it is executed.

```
VSA II Utility v3.02
Copyright (c) 1999-2003 by Advanced Micro Devices

VSM Type      Event                Parameters
--------    -----------    ----------------------------------------
SysMgr      I/O Trap       Ports = 9D00-9D7F
            I/O Trap       Ports = 9C00-9C0F
            I/O Trap       Ports = 9C14-9C1F
            PCI Trap       08xx
            PCI Trap       78xx
            Virt. Reg.     Class = Miscellaneous [00:08]
            A20 & Reset

            Legacy      I/O Trap       Port = 000B (writes)

            PME            Pin 28 (level; IRQ)
            GPIO           Pin 0  (level; INTA#)
            GPIO           Pin 7  (level; INTB#)
            GPIO           Pin 12 (level; INTC#)
            GPIO           Pin 13 (level; INTD#)
            PCI Trap       7A20 (writes)
            PCI Trap       79xx
            PCI Trap       7A40
            PCI Trap       785C-5D
            Virt. Reg.     Class = Miscellaneous [09:0A]
            Virt. Reg.     Class = SysInfo
            Virt. Reg.     Class = PM [08:0B]
            Virt. Reg.     Class = Chipset
            BLOCKIO

            SoftVG      Graphics
            PCI Trap       09xx
            Virt. Reg.     Class = VGA

OHCI        USB
            Virt. Reg.     Class = OHCI

8042        KEL
            PCI Trap       7C10 (writes)
            Virt. Reg.     Class = Keyboard

ACPI        I/O Trap       Ports = 9C20-9C3F
            PCI Trap       7C10 (writes)
            PCI Trap       7D10 (writes)
            Virt. Reg.     Class = V-ACPI
            Virt. Reg.     Class = PM

PM          Virt. Reg.     Class = PM

APM         S/W SMI        Code = 53xx
            Virt. Reg.     Class = APM

Save2Dsk    Virt. Reg.     Class = SaveToRAM

OHCI        USB

           Virt. Reg.     Class = OHCI
```

7.4 Statistics

Using the INFO /s option (lower case s) displays information about which SMIs are occurring and how much time each category of SMI requires. Each display is cumulative until the /S option (upper case S) is used to clear statistics.

```
VSA II Utility v3.02
Copyright (c) 1999-2003 by Advanced Micro Devices

VSA II STATISTICS

-----------------------------------------------------

     706 - Number of SMIs
  18,960 - Avg. clocks / SMI           (56.77 µs/SMI)
    1.1% - Total VSA overhead
           96.0% - System Manager      (54.50 µs/SMI)
            3.7% - SoftVGA             (13.40 µs/SMI)
              .3% - Legacy             (61.16 µs/SMI)
```

7.5 Event History

Event history can be displayed with the /h or /H (to clear event buffer) options. Because of the extra overhead involved, this feature is normally not enabled. A history enabled version of the system manager must be used.

7.6 Error Log

Errors can be displayed with the /e or /E (to clear logged errors) options.

7.7 MSRs

MSRs can be displayed with the /D option. One or more optional filters may be specified:

m = Display memory descriptors
i = Display I/O descriptors
l = Display LBARs
r = Display RCONFs
t = Display I/O descriptors that generate SMIs
f = Display unused descriptors

Sample Display for INFO /Di

```
VSA II Utility v3.02
Copyright (c) 1999-2003 by Advanced Micro Devices
```

MSR	High	Low	Type	Description	
100000E0	80000000	3C0FFFF0	IOD_BM	03C0-03CF	(VG)
100000E1	80000000	3D0FFFF0	IOD_BM	03D0-03DF	(VG)
100000E3	00000000	F030AC18	IOD_SC	AC1C-AC1F	(SMI)
400000E0	20000000	3C0FFFF0	IOD_BM	03C0-03CF	(GLIU0)
400000E1	20000000	3D0FFFF0	IOD_BM	03D0-03DF	(GLIU0)
400000E3	A0000000	033000F0	IOD_SC	00F0-00F1	(FG)
510200E0	60000000	1F0FFFF8	IOD_BM	01F0-01F7	(ATA)
510200E1	A000000E	F00FFF80	IOD_BM	EF00-EF7F	(AC97)
510200E2	00000009	D00FFF80	OD_BM	9D00-9D7F	(SMI)
510200E3	00000009	C00FFFE0	IOD_BM	9C00-9C1F	(SMI)
510200E4	6000000E	FF0FFFF0	IOD_BM	EFF0-EFFF	(ATA)
510200E5	00000000	3F8FFFF8	IOD_BM	03F8-03FF	(SMI)
510200E6	00000000	378FFFF8	IOD_BM	0378-037F	(SMI)
510200EA	60000000	403003F0	IOD_SC	03F6	(ATA)

Appendix

A.1 Events

Event	Meaning
EVENT_GRAPHICS	A graphics event has occurred
EVENT_USB	A USB event has occurred
EVENT_KEL	A keyboard emulation event has occurred
EVENT_ACPI	A write to ACPI register PM1_CNT has occurred
EVENT_IO_TRAP	An access to trapped I/O location has occurred
EVENT_IO_TIMEOUT	I/O inactivity timeout
EVENT_PME	A power management event has occurred
EVENT_GPIO	GPIO transition occurred
EVENT_SOFTWARE_SMI	Software SMI
EVENT_PCI_TRAP	An access to a trapped PCI configuration register has occurred
EVENT_VIRTUAL_REGISTER	An access to a virtual register has occurred
EVENT_TIMER	A timer interval has expired
EVENT_A20	A20 mask has toggled

A.2 Sample VSM Message Handler

```
#include VSA2.H
#include VR.H
unsigned long Param[MAX_MSG_PARAM];
unsigned short Messsage, Event;
unsigned short VirtualRegs[MAX_PM+1];
unsigned char Index;
unsigned long ClocksPerMs;
Hardware SystemInfo;
void VSM_msg_loop(void)
{
 // Get information about this system
 SYS_GET_SYSTEM_INFO(&SystemInfo);
 ClocksPerMs = SystemInfo.CPU_MHz * 1000L;

 do {
```

```
    // Get next message for this VSM
     Message= GET_NEXT_MSG(&Param);
     switch (Message) {
        case MSG_INITIALIZE:
          switch (Param[0]) {
               case EARLY_INIT:
                  // Early: Register as handler of Power Management virtual registers
                  SYS_REGISTER_EVENT(EVENT_VIRTUAL_REGISTER, VRC_PM, 0, 0);
                 break;

               case EARLY_INIT:
                 // Late: Register as handler of Standby button
                 SYS_REGISTER_EVENT(EVENT_GPIO, GPIO0, FALLING_EDGE, 0);
                 break;
           }
           break;

        case MSG_EVENT:
            Event = (EVENT)Param[0];
            switch (Event) {
                 case EVENT_VIRTUAL_REGISTER:
                      Index = (unsigned char) Param[1];
                      if (Index < MAX_PM) {
                          if (Param[2] = = 0) {
                             // Virtual register READ. Return register value.
                             SYS_RETURN_RESULT(VirtualRegs[Index]);
                          } else {
                             // Virtual register WRITE. Record register value.
                             VirtualRegs[Index] = (unsigned short) Param[3];
                      }
                      break;
                 case EVENT_GPIO:
                      HandleStandbyButton( );
                      break;

            } // end switch(Event)
            break;
     } // end switch(Message)
  } while (1);
}
```

A.3 Virtual Registers Classes and Index IDs

These definitions are found in the VR.H source file.

```
#define VRC_MISCELLANEOUS       0x00      // Miscellaneous Class
     #define VSA_VERSION_NUM    0x00
     #define HIGH_MEM_ACCESS    0x01
     #define GET_VSM_INFO       0x02
     #define SIGNATURE          0x03
     #define CTRL_ALT_DEL       0x06
     #define MSR_ACCESS         0x07
     #define PCI_INT_AB         0x09      // GPIO pins for INTA# and INTB#
     #define PCI_INT_CD         0x0A      // GPIO pins for INTC# and INTD#

#define VRC_VG                  0x02      // SoftVG Class
     #define VG_MEM_SIZE        0x00      // 512K units
     #define VG_FP_TYPE         0x02      // Flat panel type data
     #define VG_FP_OPTION       0x03      // Flat panel option data

#define VRC_APM                 0x03      // APM Class
     #define REPORT_EVENT       0x00
     #define CAPABILITIES       0x01
     #define APM_PRESENT        0x02

#define VRC_PM                  0x04      // Legacy PM Class
     #define PM_MODE            0x00
     #define POWER_STATE        0x01
     #define DOZE_TIMEOUT       0x02
     #define STANDBY_TIMEOUT    0x03
     #define SUSPEND_TIMEOUT    0x04
     #define PS2_TIMEOUT        0x05
     #define RESUME_ON_RING     0x06
     #define VIDEO_TIMEOUT      0x07
     #define DISK_TIMEOUT       0x08
     #define FLOPPY_TIMEOUT     0x09
     #define SERIAL_TIMEOUT     0x0A
     #define PARALLEL_TIMEOUT   0x0B
     #define IRQ_WAKEUP_MASK    0x0C
     #define SUSPEND_MODULATION 0x0D
     #define SLEEP_PIN          0x0E
     #define SLEEP_PIN_ATTR     0x0F
     #define SMI_WAKEUP_MASK    0x10
     #define INACTIVITY_CONTROL 0x11
     #define PM_S1_CLOCKS       0x12

#define VRC_INFRARED            0x05      // Infrared Class

#define VRC_TV                  0x06      // TV Encoder Class
     #define TV_ENCODER_TYPE    0x00
     #define TV_CALLBACK_MASK   0x01
     #define TV_MODE            0x02
     #define TV_POSITION        0x03
     #define TV_BRIGHTNESS      0x04
     #define TV_CONTRAST        0x05
     #define TV_OUTPUT          0x06
     #define TV_TIMING          0x10

#define VRC_ACPI                0x08      // ACPI Class
     #define ENABLE_ACPI        0x00      // Enable ACPI Mode
     #define SCI_IRQ            0x01      // Set the IRQ the SCI is mapped to
     #define ACPINVS_LO         0x02      // new calls to send 32-bit phys Address of
     #define ACPINVS_HI         0x03      // ACPI NVS region to VSA
     #define GLOBAL_LOCK        0x04      // read requests semaphore, write clears
     #define WAKE_FLAG          0x05      // Read, Write to Clear, indicates system woke
```

```
     #define RW_PIRQ            0x06      // read/write PCI IRQ router regs in SB F0
     #define SLPB_CLEAR         0x07      // clear sleep button GPIO status's
     #define PIRQ_ROUTING       0x08      // Read PCI IRQ routing based on BIOS setup
     #define ACPI_UNUSED2       0x09
     #define ACPI_UNUSED3       0x0A
     #define PIC_INTERRUPT      0x0B
     #define ACPI_PRESENT       0x0C

#define VRC_ACPI_OEM            0x09      // OEM ACPI Class

#define VRC_POWER               0x0A      // Power Controller Class
     #define BATTERY_UNITS      0x00
     #define BATTERY_SELECT     0x01
     #define AC_STATUS          0x02
     #define BATTERY_SELECT     0x03
     #define BATTERY_STATUS     0x04
     #define BATTERY_PERCENTAGE 0x05
     #define BATTERY_TIME       0x06

#define VRC_OHCI                0x0B      // OHCI Class
     #define SET_LED            0x00
     #define INIT_OHCI          0x01

#define VRC_KEYBOARD            0x0C      // Keyboard Emulation Class
     #define KEYBOARD_PRESENT   0x00
     #define SCANCODE           0x01
     #define MOUSE_PRESENT      0x02
     #define MOUSE_BUTTONS      0x03
     #define MOUSE_XY           0x04

#define VRC_DDC                 0x0D      // Video DDC Class
     #define VRC_DDC_ENABLE     0x00
     #define VRC_DDC_IO         0x01

#define VRC_STR                 0x0F      // Virtual Register class
     #define RESTORE_ADDR       0x00      // Physical address of MSR restore table
```

A.4 Document Revision History

This section reports the revision/creation process of the porting guide. Any revisions (i.e., additions, deletions, parameter corrections, etc.) are recorded in the table below.

Table A-1. Revision History

Revision # (PDF Date)	Revisions / Comments
A (21-Feb-2005)	Initial release.
B (23-Mar-2006)	Removed “Confidential”.

www.amd.com

One AMD Place • P.O. Box 3453 • Sunnyvale, CA 94088-3453 USA • Tel: 408-749-4000 or 800-538-8450 • TWX: 910-339-9280 • TELEX: 34-6306